Normalizing Flows for Implicit Bayesian Neural Networks

Student: Weijiang Xiong Supervisor: Markus Heinonen

1 Introduction and Motivation

1.1 Types of Uncertainties in Deep Learning

Deep learning methods have led to significant advances in research on Artificial Intelligence. Typically, we will create a model $\mathcal{M}$ to capture the patterns in a given dataset $\mathcal{D}$. More specifically, for supervised learning tasks, the dataset usually consists of pairs of inputs and labels $(\mathbf{X}, \mathbf{y})$. Thus, the problem can be described as: choosing an optimal set of parameters $\mathbf{w}$ for the model $\mathcal{M}$, so the model prediction $\mathcal{M}_{\mathbf{w}}(\mathbf{X})$ is closest to the label $\mathbf{y}$ under a distance measurement (loss function) $\mathcal{L}$.

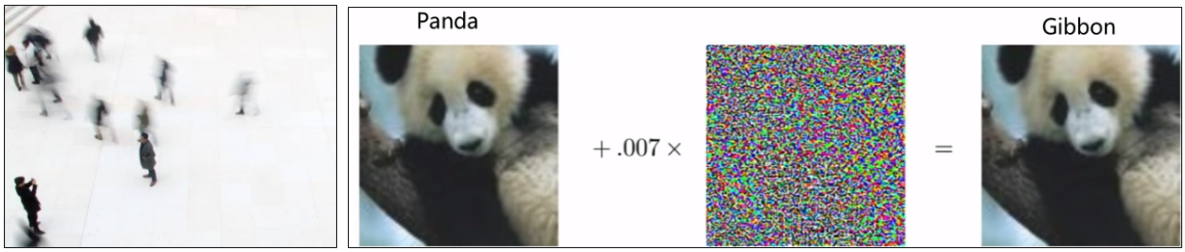
While the optimizing problem of parameter $\mathbf{w}$ can be tackled by error backpropagation and gradient-based approaches, such as ADAM [1], there is no guarantee of an optimal parameter set $\mathbf{w}$. On the contrary, in practice, we will randomly initialize $\mathbf{w}$ and then start an learning algorithm. And it's often the case that the learning algorithm behaves stably, which means if we run the same training program multiple times, the different obtained models will have comparable training losses after several epochs, and also similar accuracies on test data. Those models have different initial parameters, and will end up having different weights after learning process.

A question that rises is how to make use of those models. In practice, we often pick the one with highest test accuracy, even if those models have almost equal performance. However, with the accuracy on the limited testing data, we are not sure whether the picked model is the *truly best* model that correctly models the true data-generating process. Therefore, in Bayesian Deep Learning, we model this uncertainty with a probability distribution over the weights $p(\mathbf{w}|\mathcal{D})$, which is known as *epistemic uncertainty* or simply *weight uncertainty*. And this kind of neural network is known as Bayesian Neural Network (BNN), which marginalize over the weight posterior to provide predictive distributions [2]:

$$p(\mathbf{y} \mid \mathbf{X}, \mathcal{D}) = \int p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) p(\mathbf{w} \mid \mathcal{D}) d\mathbf{w}. \tag{1}$$

As we obtain more data, the results on the dataset will be more reliable, and thus weight uncertainty could be explained away with enough data [3].

On the other hand, real-world data are often captured by sensors, for example, images are taken by cameras and audio waves are recorded by microphones. As a result, sensor noise and motion noise will lead to uncertainties in recorded data. Figure 1 (a) shows an example of motion blur, where the images of some moving people become unclear. This kind of uncertainty is known as *aleatoric uncertainty* or simply *input uncertainty*. Unfortunately, aleatoric uncertainty is inherent in data, and can not be explained away even if we obtain more data [3].



(a) Motion Blur (b) Adversarial Attack

Figure 1

1.2 Motivation for Modelling Uncertainties

In this project, we study the uncertainty modelling problem in the context of image classification, a basic task in computer vision. The motivation comes from two problems in deep neural networks.

First, conventional DNNs are considered to be deterministic. Because they only learn a single set of parameters, and thus providing a point estimation. However, from the discussions above, we see that uncertainty exists in both the network weight and input data. Moreover, in applications that involves decision making (such as autonomous driving), we would like to consider all probable results and make sure the decision won't violate any rules or hurt anybody in all these conditions. Thus, uncertainties must be seriously taken into account, and a point estimation won't be sufficient.

Second, DNNs are usually overconfident about its results [4], even if the results are wrong. Figure 1 (b) shows an adversarial example [5], where we mix a carefully-chosen noise image to the image of a panda. As a result, the network classifies the mixed image as a gibbon with very high confidence. This error-prone overconfidence makes the output of DNNs even less reliable. Therefore, we would like to learn a model that is *well-calibrated*, which means it's confidence should match the actual accuracy. Then the distribution of confidence will reflect the uncertainties of the prediction.

In short, an ideal model should learn the uncertainties its task and provide trustworthy confidence.

1.3 Project Idea

In theory, we may consider both epistemic uncertainty and aleatoric uncertainty, but previous research has shown that for computer vision tasks, it's more efficient to model input uncertainty rather than weight uncertainty [3, 6]. For a deep neural network with millions of parameters, placing a distribution on weights is computationally expensive, but the input space makes a more suitable option. Because the size of an image or a feature map is usually much smaller than the network itself.

Trinh et al. [6] proposed an implicit Bayesian Neural Network (iBNN) that learns a Gaussian distribution for the input noise in each network layer. Compared to conventional neural networks, iBNN achieves higher classification accuracy and is also better calibrated. Although the variance of Gaussian distribution provides a reasonable estimation for uncertainties, real-world image data have various contents and the true input uncertainty could have more complicated distributions. Therefore, in this project, we will further extend the iBNN with Normalizing Flows [7] to support more flexible distributions.

2 Normalizing Flows for Implicit BNN

2.1 General Network Structure

The general structure of iBNN-style network is similar to conventional deep neural network. As shown in Figure 2, an iBNN-style network contains a series of deterministic layers and stochastic layers. These two kinds of layers have the same interface, i.e., the same format for input and output, but different internal mechanisms. A deterministic layer refers to the building blocks of conventional neural network, such as convolution layer and linear layer. Meanwhile, A stochastic layer extends a deterministic layer with a stochastic branch that offers a random vector to augment the original input. The workflow can also be expressed with the following equations.

$$\begin{aligned} \mathbf{f}_l(\mathbf{x}) &= layer_l\left(\mathbf{z}_l \circ \mathbf{f}_{l-1}\right) \quad l \geq 1 \\ \mathbf{f}_0 &= \mathbf{x} \\ \mathbf{z}_l &= NF_l(\mathbf{z}_0) \\ \mathbf{z}_0 &\sim q_0(\mathbf{z}) \end{aligned} \tag{2}$$

In the forward pass of a stochastic layer (indexed by l), we first sample a (set of) stochastic vector(s)

$\mathbf{z}_0$ from the base distribution $q_0(\mathbf{z})$ (Base Dist.). Then, the base samples $\mathbf{z}_0$ will go through a stack of Normalizing Flows (NF), which apply a series of invertible transforms to the input samples and output transformed samples $\mathbf{z}_l$. After that, the transformed samples will augment the output of the previous layer, i.e. $\mathbf{f}_{l-1}$, through element-wise multiplication. Finally, we put the augmented input $\mathbf{z}_l \circ \mathbf{f}_{l-1}$ into the deterministic component (Det. Compo) of that stochastic layer, which is literally a deterministic layer.

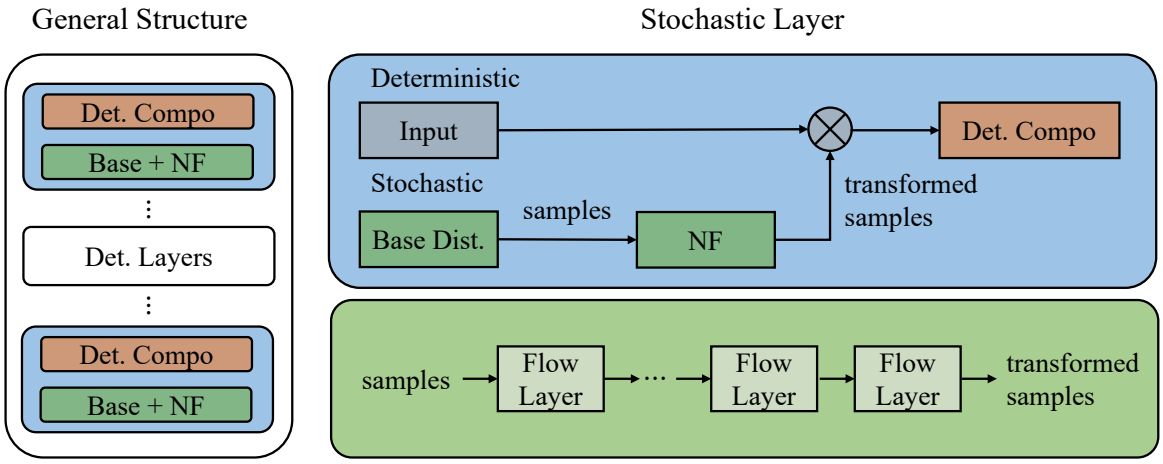


Figure 2: Network Structure

Notably, we assume that each layer has its own stochastic part, which is independent from other layers. Besides, the deterministic and stochastic part could be trained separately, and we can even migrate the weights for deterministic part from a pre-trained model.

2.2 Properties Normalizing Flows

In probabilistic modeling and inference, we often need to find a balance between expressiveness and tractability. While simple distributions, such as Gaussian, have nice analytical properties and is efficient to compute, their expressiveness is also highly limited. Sometimes even if we just move one step more complicated, the problem instantly lose analytical solution. For example, the KL divergence of two Gaussian distributions have closed form solution, but if we change one of them into a mixture of Gaussian, there won't be general solution anymore.

Normalizing Flows are a series of bijective (invertible) functions, with which we can construct arbitrarily complex distributions from a simple distribution. In recent years, Normalizing Flows have been successfully applied to density estimation [7] and image generation [8], where the exact value of parameters are learned from data. These research works have shown the expressiveness of NFs, and in this project, we will apply them to image classification task and learn the uncertainties of images and feature vectors.

Mathematically, a normalizing flow consists of one or several invertible transforms:

$$\begin{aligned} \mathbf{z}_k &= g_k(g_{k-1}(\ldots g_1(\mathbf{z}_0))) \\ &= g_k \circ g_{k-1} \circ \ldots \circ g_1(\mathbf{z}_0), \end{aligned} \tag{3}$$

where $g_1, \ldots, g_k$ are invertible functions, $\mathbf{z}_0$ is a continuous random variable with simple distribution (such as Gaussian) and $\mathbf{z}_k$ is the transformed random variable with a complex distribution. The key

benefit is that we can evaluate the log probability density value of $\mathbf{z}_k$ with the following equation:

$$\log q(\mathbf{z}_k) = \log \left[q_0(\mathbf{z}_0) \prod_{i=1}^{k} |\det J_{g_i}(\mathbf{z}_{i-1})|^{-1} \right] = \log q_0(\mathbf{z}_0) - \sum_{i=1}^{k} \log |\det J_{g_i}(\mathbf{z}_{i-1})| \,. \tag{4}$$

While $\log q_0(\mathbf{z}_0)$ is the log probability of the initial random variable (base distribution), the second term accounts for the transformations made by the k layers of flows. First, $\mathbf{z}_{i-1}$ means the random variable transformed by previous $i-1$ layers of flow, i.e., $\mathbf{z}_{i-1} = g_{i-1} \circ \ldots \circ g_1(\mathbf{z}_0)$. Then, $J_{g_i}(\mathbf{z}_{i-1})$ computes the Jacobian matrix of the i-th layer of flow with respect to its immediate input $\mathbf{z}_{i-1}$. After that, we compute the determinant (det) of the Jacobian matrix, take its absolute value ($|\cdot|$) and then map to log space (log). For a tutorial on normalizing flow, please refer to [9], and for detailed proof please have a look at [10].

But here is an simplified example. Consider a general function $y = f(x)$, where x is a Gaussian random variable. Then a small interval on the x axis $[x, x + dx]$ is mapped to $[y, y + dy]$. If we think of the corresponding probability mass on the two intervals like a physical flow (wind), the total probability mass will stay the same, just like the mass stays the same in a physical flow. Meanwhile, the density and volume could change because of outer environment. For wind, that would be different pressure in the atmosphere. But the product of density and volume, i.e., mass, must not change. In in our example, density becomes probability density $p(x)$ and $p(y)$, and volume becomes $|dx|$ and $|dy|$. Here we take the absolute value because volume is always positive, but $|dx|$ and $|dy|$ could be either positive or negative. Therefore we have an equation $p(x)|dx| = p(y)|dy|$, which is even simpler in log space:

$$\log p(y) = \log p(x) - \log |\frac{dy}{dx}|. \tag{5}$$

Similarly, if we have another function $z = g(y)$, then

$$\log p(z) = \log p(y) - \log |\frac{dz}{dy}|. \tag{6}$$

Adding up the two equations we know the log probability density of random variable z:

$$\log p(z) = \log p(x) - (\log |\frac{dz}{dy}| + \log |\frac{dy}{dx}|) \tag{7}$$

In fact, we know the source of z can be traced back to x, because $z = g(y) = g(f(x))$, but the distribution of z could be much more complicated than x because of the changes induced by f and g. If we generalize into vector case from scalar case, the derivatives $\frac{dz}{dy}$ and $\frac{dy}{dx}$ will also become Jacobian matrices. Then we will have Equation 4.

2.3 Examples of Flow

Although Equation 4 provides a straightforward method for density evaluation, the Jacobian determinant $\det J_{g_i}(\mathbf{z}_{i-1})$ in the expression makes it computationally expensive. Specifically, a recent research on matrix multiplication has indicated a complexity at $\mathcal{O}(n^{2.37})$ [11]. Therefore, we still need to carefully choose the functions in the flow stack to have feasible determinant computation. In this section, we will briefly introduce three kinds of flows with tractable Jacobian determinant: planar flow [7], affine coupling flow [12] and 1×1 invertible convolution [8].

Planar Flow belongs to the Residual Flow family [10], which means the transformed vector $\mathbf{z}'$ is a sum of the original vector $\mathbf{z}$ and a residual term. Equation 8 shows the transform, Jacobian matrix and the Jacobian determinant of planar flow. Notably, we need to use the Matrix Determinant Lemma

to get the final expression of Jacobian determinant.

$$\begin{aligned} \mathbf{z}' &= \mathbf{z} + \mathbf{v}\sigma\left(\mathbf{w}^\top \mathbf{z} + b\right) \\ J_g(\mathbf{z}) &= \mathbf{I} + \sigma'\left(\mathbf{w}^\top \mathbf{z} + b\right)\mathbf{v}\mathbf{w}^\top \\ \det J_g(\mathbf{z}) &= 1 + \sigma'\left(\mathbf{w}^\top \mathbf{z} + b\right)\mathbf{w}^\top \mathbf{v} \end{aligned} \tag{8}$$

Affine Coupling Flow splits the input vector into two groups, keep one group unchanged, and apply element-wise transform to the other group.

$$\begin{aligned} \mathbf{z}'_{1:d} &= \mathbf{z}_{1:d} \\ \mathbf{z}'_{d+1:D} &= \mathbf{z}_{d+1:D} \odot \exp\left(s\left(\mathbf{z}_{1:d}\right)\right) + t\left(\mathbf{z}_{1:d}\right) \end{aligned} \tag{9}$$

Here, the first d elements $\mathbf{z}_{1:d}$ belongs to a group, and the rest $\mathbf{z}_{d+1:D}$ belongs to another. Meanwhile, $s(\cdot)$ and $t(\cdot)$ can be any functions. As a result, the Jacobian matrix will be triangular (Equation 10), and thus the determinant will be the product of diagonal elements. Usually we need the log of determinant, which turns out to be the sum of elements in $s(\mathbf{z}_{1:d})$.

$$\begin{aligned} J_g(\mathbf{z})^\top &= \begin{bmatrix} \mathbf{I}_d & 0 \\ \frac{\partial \mathbf{z}'_{d+1:D}}{\partial \mathbf{z}_{1:d}} & \operatorname{diag}\left(\exp\left[s\left(\mathbf{z}_{1:d}\right)\right]\right) \end{bmatrix} \\ \log\det J_g(\mathbf{z}) &= \operatorname{sum}(s(\mathbf{z}_{1:d})) \end{aligned} \tag{10}$$

1×1 **Invertible Convolution** is specially adapted for image data, and apply a shared weight matrix $\mathbf{W}$ across the height and width dimensions. If we have an input image or feature map with size (C, H, W), the size of weight matrix will be $C \times C$. Then, for each length-C feature vector $\mathbf{z}_{i,j}$ across the height and width dimensions, we have the following equations:

$$\begin{aligned} \mathbf{z}'_{i,j} &= \mathbf{W}\mathbf{z}_{i,j} \\ J_g(\mathbf{z}_{i,j}) &= \mathbf{W} \\ \det J_g(\mathbf{z}_{i,j}) &= \det \mathbf{W} \end{aligned} \tag{11}$$

2.4 Network Optimization

For this part, we will leverage the idea of variational inference [7], which is learning a flow based distribution that best approximates the true distribution. The general problem formulation is the same as the original iBNN [6], but the flow part has made some difference in the details.

Loss Formulation

Suppose the parameters θ of the deterministic part has already been trained, or has been migrated from a pre-trained model. With a dataset $\mathcal{D}$, we now want to learn the input uncertainty of each layer, which are described by the posterior distribution of latent random variables $\mathbf{z}_1, ..., \mathbf{z}_l$. That is, we want to know the posterior distribution of the latent variables $p(\mathbf{z}_{1:l}|\mathcal{D};\theta)$, but the true model is so complicated that we decide to use a flow-based posterior $q(\mathbf{z}_{1:l};\theta)$ to approximate it. To this end, we minimize the KL divergence between the true posterior $p(\mathbf{z}_{1:l}|\mathcal{D};\theta)$ and its approximation $q(\mathbf{z}_{1:l};\theta)$:

$$\begin{aligned} KL[q(\mathbf{z}_{1:l};\theta)||p(\mathbf{z}_{1:l}|\mathcal{D};\theta)] &= \mathbb{E}_q[\ln q(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathbf{z}_{1:l}|\mathcal{D})] \quad \text{(omit } \theta \text{ for simplicity)} \\ &= \mathbb{E}_q[\ln q(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln \frac{p(\mathcal{D}|\mathbf{z}_{1:l})p(\mathbf{z}_{1:l})}{p(\mathcal{D})}] \quad \text{(Bayes's rule)} \\ &= \mathbb{E}_q[\ln q(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathbf{z}_{1:l})] + \mathbb{E}_q[\ln p(\mathcal{D})] \\ &= KL[q(\mathbf{z}_{1:l})||p(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] + \mathbb{E}_q[\ln p(\mathcal{D})]. \end{aligned} \tag{12}$$

By rearranging terms, the equation becomes

$$\mathbb{E}_q[\ln p(\mathcal{D})] = \underbrace{\mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - KL[q(\mathbf{z}_{1:l})||p(\mathbf{z}_{1:l})]}_{ELBO} + KL[q(\mathbf{z}_{1:l};\theta)||p(\mathbf{z}_{1:l}|\mathcal{D};\theta)]. \tag{13}$$

Because KL divergence is always larger than 0 and the probability of dataset is constant, minimizing the KL term is equivalent to maximizing ELBO. Since we assume different layers have independent stochastic parts, $\mathbf{z}_1, ..., \mathbf{z}_l$ are independent, then $q(\mathbf{z}_{1:l})$ could be factorized into a product $q(\mathbf{z}_{1:l}) = \prod_{i=1}^{l} q(\mathbf{z}_i)$. Further, we can decompose the second term in $ELBO$ into a sum of $\ln p(\mathbf{z}_i)$, where we use β-weighted KL term to provide a more flexible bound:

$$ELBO = \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - \beta \sum_{i=1}^{l} KL[q(\mathbf{z}_i)||p(\mathbf{z}_i)]. \tag{14}$$

In this equation, the first term represents data likelihood (averaged over samples), which can be calculated directly with the prediction results. Specifically, if we capture the likelihood with a categorical distribution, and the network outputs are processed by a Softmax function, the log likelihood is literally equal to the cross entropy loss [13]. The second term is the KL divergence between the flow based posterior and the prior, which controls the complexity of the flows.

KL Divergence Let's analyze the KL of just one layer. For simplicity, we remove the layer index and use a subscript to indicate the level of flow (k in total). Take care that the prior is chosen for the *transformed samples* $\mathbf{z}_k$ not the *base samples* $\mathbf{z}_0$, although they could be the same distribution. Then, the KL term of a layer could be expressed as follows:

$$\begin{aligned} KL[q(\mathbf{z}_k)||p(\mathbf{z}_k)] &= \mathbb{E}_q[\log q(\mathbf{z}_k)] - \mathbb{E}_q[\log p(\mathbf{z}_k)] \\ &= \mathbb{E}_{q_0}\left[\log q_0(\mathbf{z}_0) - \sum_{i=1}^{k} \log|\det J_{g_i}(\mathbf{z}_{i-1})|\right] - \mathbb{E}_{q_0}[\log p(\mathbf{z}_k)] \\ &= \mathbb{E}_{q_0}[\log q_0(\mathbf{z}_0)] - \mathbb{E}_{q_0} \sum_{i=1}^{k} \log|\det J_{g_i}(\mathbf{z}_{i-1})| - \mathbb{E}_{q_0}[\log p(\mathbf{z}_k)] \end{aligned} \tag{15}$$

We first expand the KL divergence by definition, and then plug in the property of normalizing flows in Equation 4. Meanwhile, we can use the change of variable technique to move the expectation from flow posterior q to the base distribution q_0. But in practice, they are both the arithmetic average over samples. As a result, the first term is the log probability of initial samples on the **base** distribution. The second term reflects the property of the learned flow, and can be named "log det jacobian". The third term is the log probability of transformed samples on the **prior** distribution.

While it's tempting to write the first and third term as $KL[q_0||p]$, it's actually wrong, because q_0 and p does not apply to the same random variable. When we write $KL[q_0||p]$, it in fact means, the two distributions should share the same variable $\mathbf{z}$, which is then integrated out.

$$KL[q_0||p] = \int q_0(\mathbf{z}) \log \frac{q_0(\mathbf{z})}{p(\mathbf{z})} d\mathbf{z} \tag{16}$$

However, that is not our case, because q_0 is the base distribution and applies to the base random variable $\mathbf{z}_0$, while p is placed on the transformed variable $\mathbf{z}_k$.

In summary, with Equation 14 and 15, we can add up the KL divergence layer by layer, and combine with the data likelihood term.

3 Experiments and Analysis

Expected Calibration Loss

3.1 2D Classification Example

3.2 Image Classification

4 Implementation Details

how did you code up all these stuff?

mention how I implemented 2d planar here.

also say the number of samples for training and testing

5 Discussions

unsolved problems

need a good way to make use of the distribution, right now we just take the mean and variance

how do we evaluate the quality of the posterior?

some thoughts on possible improvements

KL[q||p] = cross_entropy(q, p) - entropy(q), adding an entropy to ELBO will be equivalent to multiplying "entropy(q)" by a coefficient larger than 1

6 The first Problem

You may briefly state how you solve this problem here

6.1 The first part of the problem

in the first subsection, build a list with `\begin{rlisting}`

- just write something random
- another random line

build an ordered list with `\begin{renum}`

1. just write something random
2. another random line

6.2 The second part of the problem

in the second subsection, insert a figure

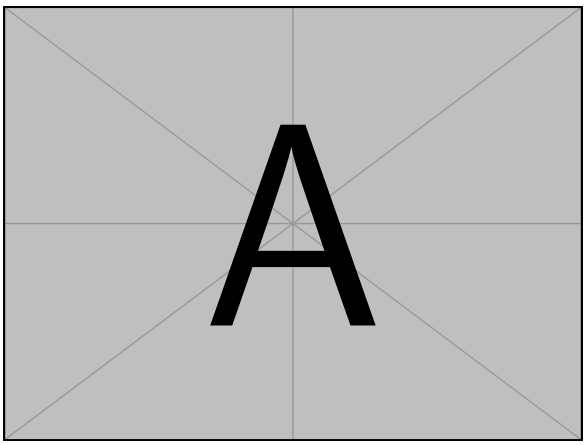
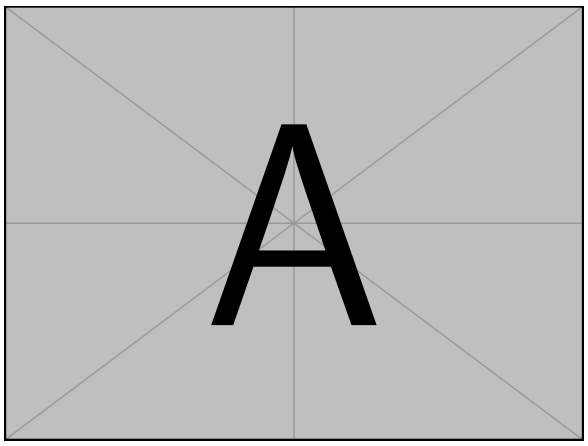
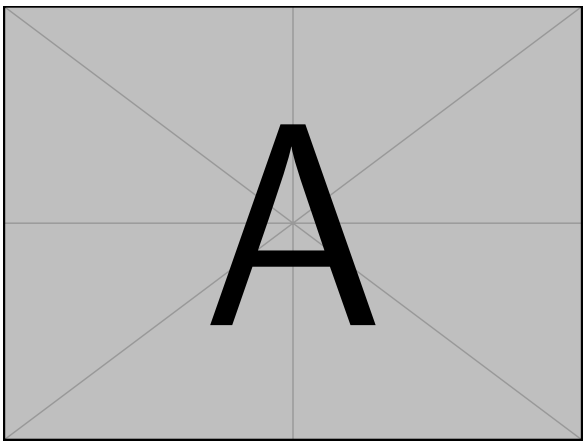


Figure 3: Add captions here

Two figures with individual numbering and caption. Comment out `\caption{}` to remove individual caption for subfigures.



(a) some comments on this figure



(b) some other comments

Figure 4: Geometrical figures

Text with picture left-right structure, remember to keep the blank line below (maybe not very useful)

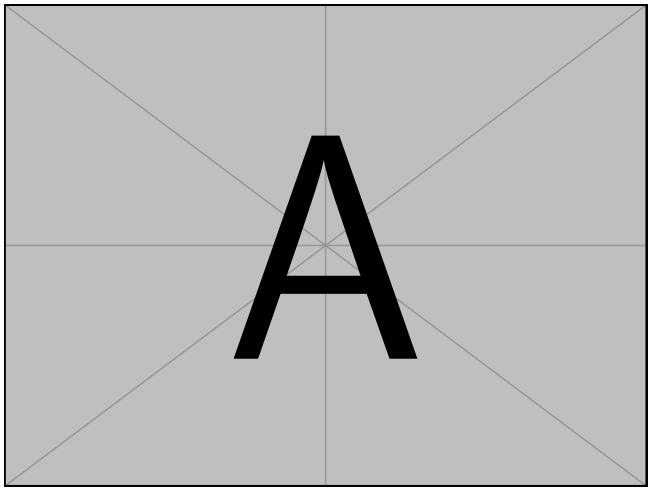


Figure 5: some comments

aa	bb
cc	dd

Table 1: Table caption

Maybe you need to cite something [14], and you can specify the IEEE style by using in the end `\bibliographystyle{ieee_fullname}`, Elsevier Style with `elsarticle-num` and Harvard Style with `elsarticle-harv0`. After this, please compile the tex file with `pdflatex->bibtex->pdflatex->pdflatex`. If the main text has not cited anything, just use `pdflatex`

Sometimes we need to start a new page with `\newpage` for a new question.

7 A new section for a new problem

7.1 the first step

this is for inserting codes with or without a bounding box, remember to use `PDFLaTeX`

```
import numpy as np
def some_function(some_variables):
    pass
# even put an under line in codes
undelrine
```

You will need to explain the codes a bit.

```
import numpy as np
def some_function(some_variables):
    pass
```

You will need to explain the codes a bit.

7.2 the second step

We may need to type matrix equations

$$T = \begin{bmatrix} m_u & 0 & 0 \\ 0 & m_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & u_0 \\ 0 & 1 & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} fm_u & 0 & u_0 \\ 0 & fm_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{17}$$

If you don't want the auto numbering

$$T = \begin{bmatrix} m_u & 0 & 0 \\ 0 & m_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & u_0 \\ 0 & 1 & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} fm_u & 0 & u_0 \\ 0 & fm_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}.$$

or build a table to present data

Table 2: Some random numbers

(α, u)	0.7	0.8	0.9	1
1	25.3183	25.5442	25.7192	25.8589
2	36.8104	37.6098	38.2343	38.7359
3	44.1152	46.6144	48.6152	50.2597
4	39.3190	44.5661	48.3255	51.3635
5	18.0562	35.4232	42.3844	47.0238

References

[1] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980.

[2] A. G. Wilson, P. Izmailov, Bayesian deep learning and a probabilistic perspective of generalization, arXiv preprint arXiv:2002.08791.

[3] A. Kendall, Y. Gal, What uncertainties do we need in bayesian deep learning for computer vision?, arXiv preprint arXiv:1703.04977.

[4] C. Guo, G. Pleiss, Y. Sun, K. Q. Weinberger, On calibration of modern neural networks, in: International Conference on Machine Learning, PMLR, 2017, pp. 1321–1330.

[5] I. Goodfellow, Adversarial examples and adversarial training, CS 231N Lecture Slides, Stanford University (2017).

[6] T. Trinh, S. Kaski, M. Heinonen, Scalable bayesian neural networks by layer-wise input augmentation, arXiv preprint arXiv:2010.13498.

[7] D. Rezende, S. Mohamed, Variational inference with normalizing flows, in: International conference on machine learning, PMLR, 2015, pp. 1530–1538.

[8] D. P. Kingma, P. Dhariwal, Glow: Generative flow with invertible 1x1 convolutions, arXiv preprint arXiv:1807.03039.

[9] E. Jang, Normalizing flows tutorial, https://blog.evjang.com/2018/01/nf1.html (2018).

[10] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, B. Lakshminarayanan, Normalizing flows for probabilistic modeling and inference, arXiv preprint arXiv:1912.02762.

[11] J. Alman, V. V. Williams, A refined laser method and faster matrix multiplication, in: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2021, pp. 522–539.

[12] L. Dinh, J. Sohl-Dickstein, S. Bengio, Density estimation using real nvp, arXiv preprint arXiv:1605.08803.

[13] F. K. Gustafsson, M. Danelljan, T. B. Schon, Evaluating scalable bayesian deep learning methods for robust computer vision, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2020, pp. 318–319.

[14] M. Danelljan, G. Häger, F. Khan, M. Felsberg, Accurate scale estimation for robust visual tracking, in: British Machine Vision Conference, Nottingham, September 1-5, 2014, BMVA Press, 2014.